Task-parallel Sparse Incomplete Cholesky Factorization using Kokkos Portable APIs.

Kyungjoo Kim  Sivasankaran Rajamanickam
H. Carter Edwards  Stephen L. Olivier  George Stelle

Collaborators : E. Boman  J. Booth  A. Bradley  C. Dohrmann  S. Hammond

Center for Computing Research, Sandia National Labs

Trilinos User Group Meeting, October 26-29, 2015
Overview

Kokkos Portable Task Parallel Programming Model

Task-parallel (In)complete Cholesky Factorization

Numerical Examples

Conclusion
Deep hierarchical features of current hardware

- Multi-socket, multi-processor, multi(or many)-core, multiple hardware threads.
- Multiple NUMA regions, multiple levels of caches, segmented and shared cache.

Need to expose more fine-grained parallelism

- Task parallelism is suitable for irregular problems: e.g., producer/consumer, recursive algorithms.
- Kokkos addresses high-level abstractions for data parallelism.
- Nested data-parallelism within a task provides better locality exploiting hardware threads.

Research highlights

- Abstractions harnessing multiple tasking backends to heterogeneous devices.
- Dependence driven asynchronous task execution with data-parallel thread team.
- Wait-free respawn task mechanism, e.g., a task on GPUs cannot wait on dependence.
- Mini-apps (e.g., sparse factorization) to support and evaluate development.
Kokkos Portable Task API

- **TaskPolicy** coordinates how and where tasks are executed, e.g., `create`, `add_dependence`, `spawn` (or `respawn`), `wait`;

- **Future** is a handle for tasks and allows dependence among them.

```cpp
void SimpleTask() {
    typedef Kokkos::Threads exec_space; // Serial, Threads, QThread
    Kokkos::TaskPolicy<exec_space> policy;
    Kokkos::Future<int> f = policy.create( Functor<exec_space>() );
    policy.spawn( f );
    Kokkos::wait( f );
}
```

- **Functor** includes a user-defined function body and associated data sets.

```cpp
class Functor<exec_space> {
public:
    Kokkos::View<exec_space> data;

    void apply( int &r_val ) {
        r_val = doSomething( data );
    }
};
```
**DAG of tasks** is implicitly formed to guide asynchronous task execution.

```cpp
void SimpleDAG() {
    typedef Kokkos::Threads exec_space;

    Kokkos::TaskPolicy<exec_space> policy;
    Kokkos::View<exec_space> x, y; // data sets for task

    auto /* future */ fx = policy.create( Functor<exec_space>( x ) );
    auto /* future */ fy = policy.create( Functor<exec_space>( y ) );

    // dependence of tasks is expressed before spawning
    policy.add_dependence( fx, fy ); // fx is scheduled after fy

    policy.spawn( fx ); // wait for now
    policy.spawn( fy ); // may immediately execute

    Kokkos::wait( policy ); // wait for all tasks to complete
}
```
Nested data parallelism with a team of threads

class Functor<exec_space> {
  public:
    Kokkos::View<exec_space> data;

    // member is mpi-like thread communicator interface
    // i.e., member.{team_rank,team_size,teamBarrier,team_reduce}
    void apply( const policy_type::member_type &member, int &r_val ) {
      Kokkos::parallel_for( TeamThreadRange( member, data.size() ),
        [&](const int i ) {
          // different indexing may be required for different
          // execution space e.g., GPU interleaved data layout
          int id = Index<exec_space>( member, i );
          doSomething( data(id) );
        } );
    }

    void SimpleTaskData() {
      typedef Kokkos::Threads exec_space;

      Kokkos::TaskPolicy<exec_space> policy;
      auto /* future */ f = policy.create_team( Functor<exec_space>() );
      policy.spawn( f );
      Kokkos::wait( f );
    }
};
Task-parallel (In)complete Cholesky Factorization

Standard procedure

1. Fill-reduced (or band-reduced) ordering: Scotch.
2. Symbolic factorization: Hysom and Pothen[1].
3. **Numeric factorization.**

Design considerations for task-parallelism

- Structure-based (in)complete factorization; fills are statically determined.
- A set of self-contained data within a task.
- Cache-friendly numeric kernels.
- Separation of concerns (concurrency is separated from parallelism):
  - Algorithm decomposes factorization into subproblems and provides dependence among them.
  - Runtime schedules tasks to parallel compute units.

→ **Objective: portable performance on most of heterogeneous architectures.**

---

1 D.Hysom and A.Pothen, Level-based incomplete LU factorization: Graph model and algorithms, 2002.
Algorithms-by-blocks

- Originally developed for distributed parallel *out-of-core* matrix computations.
- Converts basic computing units from *scalar* to *blocks*.
- Used for thread-level task parallelism[2,3]: asynchronous tasking and efficient level 3 BLAS.

```
Algorithm: A := CHOL_UNB(A)

Partition A → \( \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \)

where \( A_{TL} \) is \( 0 \times 0 \)

while \( \text{length}(A_{TL}) < \text{length}(A) \) do

Repartition

\[
\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & a_{01} & A_{02} \\ a_{10} & a_{11} & a_{12} \\ A_{20} & a_{21} & A_{22} \end{pmatrix}
\]

where \( \alpha_{11} \) is a scalar

\[
\alpha_{11} := \sqrt{\alpha_{11}}
\]

\[
a_{12}^T := a_{12}/\alpha_{11}
\]

\[
A_{22} := A_{22} - a_{12}a_{12}^T
\]

Continue with

\[
\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} A_{00} & a_{01} & A_{02} \\ a_{10}^T & \alpha_{11} & a_{12}^T \\ A_{20} & a_{21} & A_{22} \end{pmatrix}
\]

endwhile
```


Algorithms-by-blocks

- Originally developed for distributed parallel *out-of-core* matrix computations.
- Converts basic computing units from *scalar* to *blocks*.
- Used for thread-level task parallelism[2,3]: asynchronous tasking and efficient level 3 BLAS.

### Algorithm: $A := \text{CHOL\_BLK}(A)$

| Partition $A \rightarrow \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix}$ where $A_{TL}$ is $0 \times 0$ while $\text{length}(A_{TL}) < \text{length}(A)$ do Determine block size $b$
| Repartition $\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$ where $A_{11}$ is $b \times b$
| $A_{11} := \text{CHOL\_UNB}(A_{11})$
| $A_{12} := \text{TRIU}(A_{11})^{-1} A_{12}$
| $A_{22} := A_{22} - A_{11}^{T} A_{12}$
| Continue with $\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$
| endwhile

### $k$-iteration

$$
\begin{pmatrix}
\tilde{A}_{00} & \cdots & \tilde{A}_{0K} & \cdots & \tilde{A}_{0,N-1} \\
\vdots & \ddots & \vdots & \ddots & \vdots \\
\tilde{A}_{K0} & \cdots & \tilde{A}_{KK} & \cdots & \tilde{A}_{K,N-1} \\
\vdots & \ddots & \vdots & \ddots & \vdots \\
\tilde{A}_{N-1,0} & \cdots & \tilde{A}_{N-1,K} & \cdots & \tilde{A}_{N-1,N-1}
\end{pmatrix}
$$

$A_{11} := \text{CHOL}(\tilde{A}_{kk})$

$A_{12} := \tilde{A}_{kk}^{-1} (\tilde{A}_{k,k+1} \cdots \tilde{A}_{k,n-1})$

$A_{22} := (\tilde{A}_{k+1,k+1} \cdots \tilde{A}_{k+1,n-1})^{-1} (\tilde{A}_{k+1,k} \cdots \tilde{A}_{k+1,n-1})$
Algorithms-by-blocks

- Originally developed for distributed parallel \textit{out-of-core} matrix computations.
- Converts basic computing units from \textit{scalar} to \textit{blocks}.
- Used for thread-level task parallelism\cite[2,3]: \textbf{asynchronous tasking} and \textbf{efficient level 3 BLAS}.

\begin{algorithm}
\begin{align*}
\text{Algorithm: } A &:= \text{CHOL} \text{-BLK}(A) \\
\text{Partition } A &\rightarrow \begin{pmatrix}
A_{TL} & A_{TR} \\
A_{BL} & A_{BR}
\end{pmatrix} \\
\text{where } A_{TL} &\text{ is } 0 \times 0 \\
\text{while } \text{length}(A_{TL}) < \text{length}(A) \text{ do} \\
\text{Determine block size } b \\
\text{Repartition} \\
\begin{pmatrix}
A_{TL} & A_{TR} \\
A_{BL} & A_{BR}
\end{pmatrix} &\rightarrow \begin{pmatrix}
A_{00} & A_{01} & A_{02} \\
A_{10} & A_{11} & A_{12} \\
A_{20} & A_{21} & A_{22}
\end{pmatrix} \\
\text{where } A_{11} &\text{ is } b \times b \\
A_{11} &:= \text{CHOL} \text{-UNB}(A_{11}) \\
A_{12} &:= \text{TRIU}(A_{11})^{-1}A_{12} \\
A_{22} &:= A_{22} - A_{12}^{T}A_{12}
\end{align*}
\end{algorithm}

\begin{itemize}
\item \textit{k-iteration}
\end{itemize}

\begin{align*}
A_{11} &:= \text{CHOL}(\tilde{A}_{kk}) \\
A_{12} &:= \tilde{A}_{kk}^{-1}(\tilde{A}_{k,k+1} \cdots \tilde{A}_{k,n-1}) \\
A_{22} &:= \left( \begin{array}{ccc}
\tilde{A}_{k+1,k+1} & \cdots & \tilde{A}_{k+1,n-1} \\
\vdots & \ddots & \vdots \\
\tilde{A}_{n-1,k} & \cdots & \tilde{A}_{n-1,n-1}
\end{array} \right) - \left( \begin{array}{ccc}
\tilde{A}_{k+1,k} & \cdots & \tilde{A}_{k+1,n-1} \\
\vdots & \ddots & \vdots \\
\tilde{A}_{n-1,k} & \cdots & \tilde{A}_{n-1,n-1}
\end{array} \right) \\
&= \left( \begin{array}{ccc}
\tilde{A}_{k+1,k+1} - \tilde{A}_{k+1,k} \tilde{A}_{k,k+1} & \cdots & \tilde{A}_{k+1,n-1} - \tilde{A}_{k+1,k} \tilde{A}_{k,k+1} \\
\vdots & \ddots & \vdots \\
\tilde{A}_{n-1,k+1} & \cdots & \tilde{A}_{n-1,n-1} - \tilde{A}_{n-1,k} \tilde{A}_{k+1,k}
\end{array} \right)
\end{align*}

\begin{itemize}
\item Tri-Solve
\item General Matrix-Matrix mult.
\item Hermitian Rank K-update
\end{itemize}

\begin{thebibliography}{9}
\bibitem{3} A.Buttari \textit{et al.}, A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures, 2009.
\end{thebibliography}
Sparse Cholesky-by-blocks

Algorithm: $A := \text{CHOL\_BY\_BLOCKS}(A)$

Partition $A \rightarrow \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix}$

where $A_{TL}$ is $0 \times 0$

while $\text{length}(A_{TL}) < \text{length}(A)$ do

Repartition

$\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$

where $A_{11}$ is $1 \times 1$

Continue with

$\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$

endwhile

**function** genTaskChol:

Future $f = \text{create}(\text{Chol}, A_{11}(0,0))$

add_dependence($f$, $A_{11}(0,0).\text{getFuture()}$)

$A_{11}(0,0).\text{setFuture}(f)$

spawn($f$)

**function** genTaskTrsm:

for $j$ in $A_{12}.\text{nnz}()$

Future $f = \text{create}(\text{Trsm}, A_{11}(0,0), A_{12}(0,j))$

add_dependence($f$, $A_{11}(0,0).\text{getFuture()}$)

add_dependence($f$, $A_{12}(0,j).\text{getFuture()}$)

$A_{12}(0,j).\text{setFuture}(f)$

spawn($f$)

**function** genTaskHerk:

for $i$ in $A_{12}.\text{nnz}()$

if exist($A_{22}(i,j)$)

for $j$ in $A_{12}.\text{nnz}()$

Future $f = \text{create}(i=j?\text{Herk} : \text{Gemm},$

$A_{12}(0,i), A_{12}(0,j),$

$A_{22}(i,j))$

add_dependence($f$, $A_{12}(0,i).\text{getFuture()}$)

add_dependence($f$, $A_{12}(0,j).\text{getFuture()}$)

add_dependence($f$, $A_{22}(i,j).\text{getFuture()}$)

$A_{22}(i,j).\text{setFuture}(f)$

spawn($f$)

- Degree of concurrency still depends on nested dissection ordering.
- Parallelism is not strictly tied with the nested dissection tree.
- Each block records future and dependence is explicitly described from the algorithm.
Hierarchical (recursive) definition of matrices

- **CrsMatrixBase** contains sparse data arrays *i.e.*, row pointers, column indices, value array.
- **MatrixView** defines a rectangular region *i.e.*, view offsets and dimensions;
  - light-weight object with meta data only.

```
typedef CrsMatrixBase<value_type := scalar> ScalarMatrix;
typedef MatrixView<base_object := ScalarMatrix> ViewOnScalarMatrix;

typedef CrsMatrixBase<value_type := ViewOnScalarMatrix> BlockMatrix;
typedef MatrixView<base_object := BlockMatrix> ViewOnBlockMatrix;
```
2D Partitioned-block Matrix

Hierarchical (recursive) definition of matrices

- **CrsMatrixBase** contains sparse data arrays i.e., row pointers, column indices, value array.
- **MatrixView** defines a rectangular region i.e., view offsets and dimensions;
  - light-weight object with meta data only.

typedef CrsMatrixBase<value_type := scalar> ScalarMatrix;
typedef MatrixView<base_object := ScalarMatrix> ViewOnScalarMatrix;

typedef CrsMatrixBase<value_type := ViewOnScalarMatrix> BlockMatrix;
typedef MatrixView<base_object := BlockMatrix> ViewOnBlockMatrix;
Hierarchical (recursive) definition of matrices

- `CrsMatrixBase` contains sparse data arrays \textit{i.e.}, row pointers, column indices, value array.

- `MatrixView` defines a rectangular region \textit{i.e.}, view offsets and dimensions;
  - light-weight object with meta data only.

```cpp
typedef CrsMatrixBase<value_type := scalar> ScalarMatrix;
typedef MatrixView<base_object := ScalarMatrix> ViewOnScalarMatrix;

typedef CrsMatrixBase<value_type := ViewOnScalarMatrix> BlockMatrix;
typedef MatrixView<base_object := BlockMatrix> ViewOnBlockMatrix;
```
Hierarchical (recursive) definition of matrices

- **CrsMatrixBase** contains sparse data arrays *i.e.*, row pointers, column indices, value array.
- **MatrixView** defines a rectangular region *i.e.*, view offsets and dimensions;
  - light-weight object with meta data only.

```
typedef CrsMatrixBase<value_type := scalar> ScalarMatrix;
typedef MatrixView<base_object := ScalarMatrix> ViewOnScalarMatrix;

typedef CrsMatrixBase<value_type := ViewOnScalarMatrix> BlockMatrix;
typedef MatrixView<base_object := BlockMatrix> ViewOnBlockMatrix;
```
Example

A sequence of tasks generated during Cholesky-by-blocks:

(a) 1st iteration

(b) 2nd iteration

(c) 3rd iteration

(d) 4th iteration

(e) 5th iteration

Task DAG
Example from Real Problem: pwtk

- Entire task DAG is constructed to demonstrate the degree of concurrency.
- Explicit DAG is never formed and not used in task scheduling in both Pthreads and Qthreads task policies.
Numerical Examples

Test problems from UFL sparse collection

<table>
<thead>
<tr>
<th>Matrix ID</th>
<th># of rows(n)</th>
<th># of nonzeros (nnz)</th>
<th>nnz/n</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ecology2</td>
<td>999,999</td>
<td>4,995,991</td>
<td>4.99</td>
<td>Circuit theory</td>
</tr>
<tr>
<td>pwtk</td>
<td>217,918</td>
<td>11,524,432</td>
<td>52.88</td>
<td>Stiffness matrix</td>
</tr>
</tbody>
</table>

Machine specifications

<table>
<thead>
<tr>
<th>Processors</th>
<th>Intel Xeon E5-2670</th>
<th>Intel Xeon Phi</th>
<th>IBM Power8</th>
</tr>
</thead>
<tbody>
<tr>
<td># of cores</td>
<td>2x8</td>
<td>1x57</td>
<td>4x5</td>
</tr>
<tr>
<td>Clock speed</td>
<td>2.6 GHz</td>
<td>1.1 GHz</td>
<td>3.4 GHz</td>
</tr>
<tr>
<td>L2 per core</td>
<td>256 KB</td>
<td>512 KB</td>
<td>512 KB</td>
</tr>
<tr>
<td>L3</td>
<td>20 MB shared</td>
<td>-</td>
<td>8 MB per core</td>
</tr>
</tbody>
</table>

Compiler

Intel 15.2.164      GNU 4.9.2

Kokkos

- Pthreads backend with task only interface (team size = 1).
Euclid performs MPI-parallel incomplete LU.

Parallelism is extracted from 1D rowwise partition of a matrix.

Reverse Cuthill McKee (RCM) ordering is used to reduce the bandwidth of the matrix.

Bandwidth of matrices increases with an increasing level of fills.
Strong Scaling : Intel Sandy Bridge

- Speed-up = \( \frac{\text{Time for single-threaded Cholesky-by-blocks}}{\text{Time for parallel Cholesky-by-blocks}} \).
- Performance depends on matrix sparsity: ecology2 is sparser and pwtk is denser.
- With increasing fill-level, factorization is more compute-intensive.
- Tasking overhead is constant and amortized during asynchronous parallel execution.

\[
\begin{align*}
\text{ecology2} & \\
\text{pwtk} &
\end{align*}
\]
Strong Scaling : Intel Xeon Phi

- Speed-up = \frac{\text{Time for single-threaded Cholesky-by-blocks}}{\text{Time for parallel Cholesky-by-blocks}}.
- Performance depends on matrix sparsity: ecology2 is sparser and pwtk is denser.
- With increasing fill-level, factorization is more compute-intensive.
- Tasking overhead is constant and amortized during asynchronous parallel execution.

![Graphs showing speed-up for ecology2 and pwtk with varying number of threads and fill levels.](image)
Strong Scaling: IBM Power8

- Speed-up = \frac{\text{Time for single-threaded Cholesky-by-blocks}}{\text{Time for parallel Cholesky-by-blocks}}.

- Performance depends on matrix sparsity: ecology2 is sparser and pwtk is denser.

- With increasing fill-level, factorization is more compute-intensive.

- Tasking overhead is constant and amortized during asynchronous parallel execution.

![Graphs showing speed-up for ecology2 and pwtk](image)
Tasking Overhead: Intel Xeon Phi

- Efficiency = \frac{\text{Time for sequential Cholesky}}{\text{Time for single-threaded Cholesky-by-blocks}}.
- With increasing fill-level, cache-friendly sparse matrix operations on blocks exploit better data locality.
- Task granularity is problem-specific on irregular problems.

<table>
<thead>
<tr>
<th>Fill-level</th>
<th>Ecology2</th>
<th>Pwtk</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>15</td>
<td>83</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>3,071</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>105,864</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>905</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>33,182</td>
</tr>
</tbody>
</table>

Graph showing efficiency vs. fill-level for ecology2 and pwtk.
Summary

Kokkos hybrid task-data programming model

- Presented abstractions for task-data parallelism.
- Developed dependence driven task model.
- Harnessed two tasking backends: Pthreads and Qthreads.
- https://github.com/kokkos/kokkos

Task-data parallel sparse matrix factorization

- Presented sparse algorithm-by-blocks for task parallel Cholesky (in)complete factorization.
- As mini-app, provided support and feedback to design task-data interface.
- Demonstrated portable performance on multicore and manycore architectures.
- Trilinos/shylu/tacho

Kokkos tasking API and Cholesky miniapp are in the experimental phase
Ongoing and Future Work

Kokkos hybrid task-data programming model

- Asynchronous tasking on GPUs.

Task-data parallel sparse matrix factorization

- Performance optimization for task-data hybrid parallelism: e.g., algorithm design and thread team overhead.
- Supernodal direct factorization: Cholesky and LDL.
- Leverage to domain decomposition FE solver in collaboration with Clark Dohrmann (1542).